DOCUMENT RESUME

ED 364 195 IR 016 279

AUTHOR McAllister, Alan
TITLE Representing the Programming Process: Goal Structures and Action Sequences in LOGO Programming.
PUB DATE 13 Apr 93
NOTE 37p.; Paper presented at the Annual Meeting of the American Educational Research Association (Atlanta, GA, April 12-16, 1993).
PUB TYPE Reports - Research/Technical (143) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Adolescents; Elementary School Students; Foreign Countries; Intermediate Grades; Junior High Schools; Junior High School Students; Models; Preadolescents; *Problem Solving; *Programing; Research Tools; *Skill Analysis; *Thinking Skills
IDENTIFIERS *LOGO Programing Language; Strategic Thinking

ABSTRACT

This study developed representations of the LOGO programming process which provide the basis for strategy analyses and a new perspective on problem solving in complex, semantically rich task environments such as LOGO. Nine students, ages 10 to 14, who had been identified as gifted and had previous programming experience, were trained in the rudiments of LOGO. The focus of the analysis was the performance of the students on three tasks which involved graphics, word-and-list, and interactive game programming. Software tools were developed to collect and analyze the subjects' interactions with the computer while programming. The product of the analysis was a problem behavior graph which plots the programmer's path through the two problem spaces involved in LOGO programming. These graphs are the basis for identification of the programmers' strategies. Three different layers of strategies were exhibited: top-down design strategies, depth-first component base strategies, and subsidiary search and debugging strategies. The model elaborated through the strategy analysis highlights the role of the task environment in supporting the students' problem solving, and it calls into question views concerning the relationship between strategies and knowledge and the way in which goals direct the problem solving process. An illustration of an analysis of the stages in collecting and analyzing the programming process is appended. (Contains 25 references.) (Author/KRN)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it

☐ Minor changes have been made to improve reproduction quality

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy

ED 364 195

Representing the programming process: Goal structures and action sequences in LOGO programming

Alan McAllister
Brock University
April 13, 1993

Abstract

Fundamental to understanding any human activity is understanding it in terms of process, yet the representation of process poses difficulties that require fresh approaches. The study reported here develops representations of the LOGO programming process which provide the basis for strategy analyses and a new perspective on problem solving in complex, semantically rich task environments such as LOGO.

Nine students, ages ten to fourteen, who had been identified as gifted and had previous programming experience, were trained once a week for one and a half hours over a fifteen week period in the rudiments of LOGO. The focus of the analysis was the performance of the students on three tasks which involved graphics, word-and-list, and interactive game programming. To carry out the analysis, software tools were developed for the collection and analysis of subjects' interactions with the computer while programming. The product of this analysis is a representation of the programming process in the form of a problem behavior graph which plots the programmer's path through the two problem spaces which the LOGO programming environment involves. These graphs are the basis for identification of the programmers' strategies.

Three different layers of strategies were exhibited in the students' programming: top-down design strategies, depth-first component-based strategies, and subsidiary search and debugging strategies and tactics. The model elaborated through the strategy analysis highlights the role of the task environment in supporting the students' problem solving, and it calls into question commonly held views concerning the relationship between strategies and knowledge and the way in which goals direct the problem solving process.

Introduction

Fundamental to understanding any human activity is understanding it in terms of process, in terms of change, development, movement. As Vygotsky observed, "To encompass in research the process of a given thing's development in all its phases and changes ... fundamentally means to discover its nature, its essence, for 'it is only in movement that a body shows what it is'" (Vygotsky, 1978, pp. 64-65).

Yet representing process has inherent difficulties. At some level, "capturing" a process requires fixing it and making it stationary. A movie gives the illusion of life, of process, but in actuality it is simply a series of snapshots in very rapid succession. The adequacy of a representation of a process is determined, in the end, by the density of the snapshots that form

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
Alan McAllister
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

IR016279

its basis, the continuity that the representation provides for these snapshots, and the underlying structure that it reveals.

The research reported in this paper is an attempt to capture a process, specifically the programming process of a group of gifted young students learning LOGO. The representation of this process which is developed here is based on high density observations and specifies the structures of goals and action sequences that the programming process involves. It is the application of a method which follows in the process analytic tradition of Newell and Simon (1972), but which diverges from it in important respects.

The promise of the application of a new method to a phenomenon is that it will lead to new perspectives on that phenomenon. The results obtained in this research point to a view of the programming process and problem solving generally that challenges some widely held assumptions; specifically, it challenges views about the relationship between strategies and knowledge and the role of goals in problem solving. Against a background of issues of method and theory, the data sources, method, and model for this study are outlined. This is followed by a taxonomy of strategies, a report of results from the measured used, and a discussion of the issues raised in the study.

Background

Issues, both of method and theory, need to be addressed to provide a rationale for the particular approach that has been adopted here. This rationale is not designed to convince but to orient the reader to considerations that have led to the development of this approach.

In terms of method, the lack of suitable precedents in the literature for carrying out a study of the LOGO programming process suggests the need for the development of fresh approaches. In terms of theory, questions about the adequacy of problem solving models based on well-structured tasks as applied to semantically-rich, open-ended tasks suggest the need for modification of these models.

The literature on LOGO programming, especially within the educational research community, has largely focused on the issue of whether there are benefits to learning LOGO, whether the skills learned in programming have effects on other skills, particularly the core literacy and numeracy skills which are the focus of the school curriculum (Clements, 1985). However, with some notable exceptions (Carver, 1987; Carver & Klahr, 1986; Klahr & Carver, 1988), little attention has been paid to what goes on in the programming process itself (McAllister, 1991, 1992).

There is, however, a substantial literature on the programming process as exhibited by university students and professional programmers. In much of this research, some form of protocol analysis (Ericsson & Simon, 1984; Newell & Simon, 1972) is used. In this method verbalizations of problem solvers as they do tasks are used as the data base for developing and testing models of

the problem solving process. However, objections have been made in principle against the use of verbalizations (Nisbett & Wilson, 1977) and against the applicability of protocol analysis in studies of human-computer interactions (Praetorius & Duncan, 1988). In addition, there are tremendous costs and risks associated with collecting, transcribing, and coding verbalizations and making inferences from them (Waterman & Newell, 1972). Moreover, specific difficulties associated with getting children to verbalize during problem solving, may preclude the use of this approach in a study of young students' problem solving.

Models originally developed for the analysis of well-structured tasks (Newell & Simon, 1972) have been extended to semantically rich domains such as programming (McAllister, 1990). These models, which focus on the goal-directed nature of problem solving, involve a cycle of goal-setting, acting, and evaluating, with goals directing the cycle by initiating and controlling actions and serving as comparison standards for results of actions (Brenner, Ginsberg, & von Cranach, 1985). The impression left is that the goal-representations which direct action are unitary and well-articulated, and that goals are stacked or otherwise organized in a hierarchical fashion to ensure that action is under the control of one particular goal until the goal is attained or abandoned (Broadbent, 1985).

Questions can be raised about the viability of extending these models to programming. In a well-structured task, such as the Tower of Hanoi, the goal is clearly envisioned at the outset, and there is one "best" path to that goal. In contrast, while a programming task may have pre-set specifications as to what constitutes a solution, a variety of different solutions may be possible as well as different solution paths, any one of which is as satisfactory as the other. This suggests that models based on well-structured tasks may be inadequate for semantically rich domains to the extent to which they depend upon goals being well-defined.

The process analytic approach to the study of the programming process has proven fruitful, but its limitations are apparent, especially as they would apply to a study such as this. Alternative process-oriented methods do exist, however, and can provide the basis for the development of more satisfactory models of the programming process.

Within the Child Study tradition, specimen description (Barker & Wright, 1971) has been widely used to collect records of children's behavior. However, like protocol analysis, this form of naturalistic observation is extremely time-consuming and often unreliable. Automated process analysis systems, using on-line records of users' actions on computers, have been devised for mathematics tasks (Brown & Burton, 1978; Kowalski & VanLehn, 1988; Ohlsson & Langley, 1985; VanLehn & Garlick, 1987) and well-structured problem solving tasks (Smith, Smith, & Kupstas, 1991). Although it may not be feasible, at least at this point, to develop a fully automated process analysis of semantically-rich, open-ended tasks, partially automated process analysis is an achievable aim, and offers the possibility of a highly reliable method of data collection

and analysis which could be applied to large groups of subjects.

The method developed in this study is a synthesis of elements from the literature on protocol analysis, naturalistic observation, and automated user modelling. The analysis of the LOGO task environment is based on Newell and Simon's concept of problem spaces and employs automated collection and partially automated analysis of programming records to develop representations of the programming process. In what follows, the data source for this study and the method employed for developing these representations are described, and a model of the programming process is outlined and then completed using strategy analyses based on these representations.

Data source

Nine students, ages nine to fourteen (seven boys and two girls), who had been identified as gifted and had previous programming experience, were trained once a week for one and a half hours over a fifteen week period in the rudiments of LOGO, including graphics, word-and-list, and interactive game programming.

Three programming tasks were administered to the students on an individual basis. The first task required them to draw a house and a matching playhouse. This was administered to them after six weeks of training. The second task required them to write an interactive program, a High-Low number guessing game. This was administered after thirteen weeks of training. The final task was administered after the conclusion of the training and required them to program another interactive program, a Hangman game. The students were given 90 minutes for the first two tasks, and three hours for the final task. (Figure 1 shows simplified versions of programs and their outputs.) The representations of the programming process and the strategy analysis are based on records of the students' performance on these tasks.

In addition to the programming measures, three measures of programming knowledge were administered in a regular training session. The first two in-class measures were paper-and-pencil tests. The first in-class measure covered graphics knowledge and was administered the week following the individually administered graphics programming task. The second covered word-and-list commands and techniques for interactive games and was administered the week following the administration of the High-Low task. The last measure was a mixed paper-and-pencil and computer task in which the students had to debug programs, show what programs did, and write programs. The content of this measure included graphics, word-and-list commands and techniques for interactive games; students were told to do as much as they could using paper-and-pencil, and they were then allowed to use the computer to complete the problems. This measure was given in the last training session and prior to the administration of the final individually administered programming task.

Task analysis and software tools

The task analysis which provides the framework for this analysis of the programming process focuses on certain invariants of the LOGO programming environment (specifically, the IBM LOGO used in this study). These invariants provide the points of reference for the software tools that are used to develop a representation of the programming process. This discussion is designed to conceptualize how this representation is developed; a more detailed description of these tools and how they are employed is to be found in Appendix A.

LOGO is an interpreted language and highly interactive. Programming in LOGO involves writing procedures; within this environment, procedures are developed in either a define mode (line-by-line) or an edit mode (full-screen), and instructions are carried out in an immediate mode. Programming in this environment can be conceptualized as a search in two basic problem spaces, a primary problem space (the "procedural space"), which is linked with the define and edit modes in which procedures are developed, and a secondary space (the "trial space"), which is linked with the immediate mode in which experimentation with instructions and testing and debugging of procedures take place. Figure 2 is a broad characterization of the two problem spaces in the LOGO environment.

<Insert Figure 2 about here>

The recording software tool developed for this study collects a time-stamped series of snapshots of the contents of the define and edit modes when they are exited and snapshots of the immediate mode when a set of instructions is invoked; these snapshots are taken to represent states of the two problem spaces. The record generated by this tool provides the primary data base for the analysis of the programming process.

Another software tool encodes the snapshot record of states by identifying operators that change these states. Just as there are clear referents for the states of these problem spaces in the LOGO task environment, there are clear referents for the operators which change these states. In the define and edit modes, states are changed by virtue of the definition of procedures and the insertion and deletion of instructions. The encoding program classifies the operators into these categories; for the procedural space it does this by comparing past and present versions of procedures and identifying the changes that take place. This program produces an encoded record, essentially a sequence of states and operators, which is taken to represent the subject's path through the two problem spaces.

With the identification of referents for the states and operators of the two problem spaces, the structure of the task environment is fairly clear. According to this account, search in these two spaces involves going back and forth between them-- writing procedures, trying them out, and using the feedback that the computer gives to make changes to the procedures. What moves the problem solution forward are the concrete actions that the programmer takes within the environment to create, modify, and invoke

instructions in the LOGO language.

<Insert Figure 3 about here>

Figure 3 shows in an abstract form the various elements that make up the representation of the programming process. In addition to those that have already been distinguished, two other elements are necessary for representing the structure and dynamics of the process: functions and episodes.

The operators evident in the record have functions. For instance, when a programmer appends a set of instructions to draw a triangle to draw a roof, the function of the operator is identified as drawing a roof. This is clearly related to drawing a house, so the higher level function of drawing the rectangle is drawing the house. Figure 3 shows operators and functions and how the functions relate to a hierarchy of higher level functions. These hierarchical structures are assumed to reflect the programmer's goals: When a programmer inserts instructions to draw a triangle to draw a roof, it can be assumed that drawing a roof was the programmer's goal and that this goal was a subgoal of the goal of drawing a house.

Another software tool is used to assist the researcher in grouping operators according to their functions and, where appropriate, to assign these functions to more general functions. Once the operators have been assigned functions, a hierarchically structured functional record is computer-generated; this functional record is assumed to track the programmer's goal commitments and their relationships to one another.

The final step in the development of the problem behavior graph is to "episode" (Barker & Wright, 1971) the functional record. Episodes refer to beginning-to-end action sequences in the record directed at a specific goal. These episodes can have a variety of relationships to one another; the brackets in Figure 3 represent an episodal structure which occurs quite frequently in the records. (The different types of episodal structures are more fully described in Appendix A.)

In this particular episodal structure, there is a shift back and forth from one problem state to another; a change is made to a procedure, the procedure is tested, and another change is made, and so on. As can be seen in the figure, the instructions for the body of the house are written and then tested. Then a triangle is programmed; however, when the procedure is tested, the house is inverted and does not look like the roof it was intended to be. The programmer then writes instructions to invert the house and tests the procedure again. These interlinked episodes reveal the structure in the sequence of these actions.

These are the essential elements necessary for representing both the structure and dynamics of the programming process. This representation involves two kinds of structures, a hierarchical functional structure which reflects the goal representations which guide the programming process, and a sequential

structure which reflects the actual beginning-to-end actions in the record.

<Insert Figure 4 about here>

Figure 4 shows an actual problem behavior graph for the initial segment of a session in which a student was working on the first programming task. On the left, the states of the procedural space are represented, and on the right, the states of the trial space. In the middle is the problem behavior graph itself. The operators are represented as are the functions, and the indentations show the hierarchical relations between the functions. The episode brackets on the left of the middle panel show the sequential structure of the actions.

It is the analysis of these problem behavior graphs which yields the strategy analysis. But interpretation of these graphs also requires an initial model of the programming process which the strategy analysis itself elaborates and refines.

The model

The model developed for this analysis is referred to as the agenda model. It, like the models developed for well-structured tasks, is based on concept of a search control cycle. However, the way in which goals direct the cycle is viewed quite differently.

In this model, the programmer is thought to begin with a set of task instructions and ends with a completed program. What happens in between those two points is a cyclical process of search in the two problem spaces of the task environment.

<Insert Figure 5 about here>

The Figure 5 shows what happens in this cycle. When engaged in the task, the programmer represents the problem on the basis of the task instructions; this problem representation includes the goals set by the programmer, and it is updated throughout the process. In the cycle, a comparison is made between the state of the problem and the programmer's goals, goals and methods for fulfilling them are selected, operators are implemented, and actions are taken on the system. These actions bring about changes in the system which provide feedback to the programmer, and the cycle begins again with a comparison between the programmer's goals and the state of the problem. Throughout this process knowledge is being both developed and deployed, deployed when methods are retrieved and used, developed when feedback from the system is used to obtain information for achieving goals.

In representing the problem, the programmer is seen as constructing an agenda of goals initially, and this agenda is constantly updated and changed as the programmer proceeds through the task. The process can be thought of as goal-directed in much the same way that an agenda is used to manage a meeting that

can be unruly and unpredictable at times. Goals are set and some of these goals are accomplished, but there has to be flexibility in the process which allows shifts in topic, new business, and meeting unpredictable demands that arise.

In this model, strategies are presumed to control the problem solving process by organizing the decomposition and scheduling of goals and coordinating the deployment and development of knowledge in the service of attaining those goals. The categorization of strategies is done on the basis of an examination of the overall structure of the episodes, the scope and sequence of the functional elements within the episodes, and the ways in which the two problem spaces are used to develop the program.

Types of strategies

Table 1 shows the levels of strategies which are distinguished based on an analysis of the students' records. Three levels of strategies are distinguished: depth-first, component-based strategies, strategies which are top-down and breadth-first to some degree or other and are used primarily in the design phase of the programming process, and subsidiary strategies which occur within the context of these higher level strategies and are used predominately to develop knowledge within the programming process itself.

<Insert Table 1 about here>

The depth-first strategies are used after a decomposition of the problem into identifiable components. Three of these strategies are distinguished: an incremental strategy, a refinement strategy, and a modular strategy. These strategies are distinguished primarily on the basis of the way in which the components themselves are decomposed and their elements dealt with sequentially.

In the incremental strategy, a component of the program is developed sequentially, a bit at a time. To construct a square, for instance, a set of instructions to draw a line is added to the procedure and the procedure is tested. If the procedure works as intended, the programmer goes on; if not, the procedure is altered and tested again. In this way, the procedure is added to until it successfully draws the square.

In the refinement strategy, the component is developed as a whole and then the solution debugged and refined. Instead of the incremental, bit by bit approach to drawing the square, for instance, the programmer would write all the instructions necessary to draw the square, test the procedure and then refine it until it worked as intended.

In the modular strategy, the component is broken down into subunits which can be separately developed and integrated into a whole. For instance, to draw a house with a roof, the programmer would construct a procedure which had as subcomponents units to draw a square and the triangle. Typically these units

would be constructed separately and tested, and then integrated into a whole later.

In the top-down strategies the problem is not decomposed into component elements which are then worked on separately; in these strategies, at least initially, an attempt is made to deal with the whole problem at once. These strategies differ from one another in the degree to which the solution is articulated at the outset. Four major top-down strategies are distinguished: a breadth-first refinement strategy, a nominal strategy, a stratification strategy, and a sequential strategy.

In the breadth-first refinement strategy, the program is developed as a whole at all levels in an extended episode ending with a test of the superprocedure and subsequent debugging of errors in the superprocedure and its subprocedures.

The nominal strategy is heterogeneous in nature, top-down and breadth-first to a point and then depth-first thereafter. In this strategy, the programmer develops the program only at the abstract level of procedure calls, but once the level of specific instructions (involving the use of primitives) is reached, a switch is made to constructing the components in a depth-first fashion.

The stratification strategy is another self-limiting strategy. In this strategy, the programmer typically defines a superprocedure, and then codes at the next lower level; when a function is completely coded, the program is tested and debugged. The principle behind this strategy appears to be to construct a program to the point where it is possible to get some meaningful feedback which can be used to develop the program.

The last top-down strategy, the sequential strategy, is similar to the stratification strategy, but in this strategy what gets coded is determined by the ordering of the task instructions rather than the requirements of the functioning of the evolving program. Typically, the program is developed in the form of a superprocedure with the calls to procedures which follow the order found in the task instructions. Then the component procedures are defined in the same order. The flaws of this strategy become quite evident in programming an interactive game in which a looping structure is required.

The subsidiary strategies occur within the context of these major strategies. They can be divided into strategies that are anticipatory and those that are used for debugging. The anticipatory strategies of incorporative search and inferential search are employed to gather information in the trial space for use in developing procedures. In incorporative search, the successful instructions are incorporated into the procedures, whereas in inferential search, inferences made on the basis of experiments in the trial space are used in the development of the procedures.

The debugging strategies are used in the context of seeking feedback from the

trial space. The reactive debugging strategies use tests of procedures to adjust and refine the procedures. The kind of debugging depends upon the nature of the feedback received. When the output of the procedure is discrepant with what the programmer intended, adjustments are made to the procedure. When the system gives explicit error messages, the error messages can be used as diagnostic of the error or simply as symptomatic of the error. Depending upon the programmer's interpretation, the response can be repair of the problem indicated by message, repair of a problem at another level which is presumed to be the source of the error, or further exploration to find the source of the error.

The strategies of verification search and isolation search are specific tactics associated with debugging. Verification search is used to verify that a procedure was invoked as intended. Isolation search is used when a component of the program interacts with other components and is isolated so that it can be debugged.

For the purposes of identifying the strategies within the students' records, the sessions were divided into major episodes, defined as enclosing episodes which have a set of related goals. These sets of goals might be related to a single component of the program, for instance, the roof of the house in the first task, or to working out a specific aspect of the program, such as the flow of control in one of the interactive games. Figure 6 shows an example of how the strategies are identified for a set of students; the example is taken from the third task in which all three levels of strategies were exhibited.

<Insert Figure 6 about here>

Results of measures

The results from the three individually administered programming tasks and the measures of programming knowledge are shown in Table 2. In the case of the programming tasks, the measure of effectiveness was the number of components of the program which were successfully implemented, and in the case of the in-class measures, the level of knowledge was determined by scoring the number of problems correctly solved. Table 2 shows the students in rank order for both these measures, and shows the two highest level strategies that they used. Table 3 shows the intercorrelations among the measures; significant correlations (using a generous $p > .10$ level) are shown with an asterix.

<Insert Table 2 about here>
<Insert Table 3 about here>

Although the first programming task and the first measure of LOGO knowledge did not significantly correlate with one another, the measures within the other two sets did. However, the measures of knowledge did not significantly correlate with one another, although the second and third programming tasks did, and the third programming task significantly correlated with all three of

the in-class measures of knowledge.

A few students showed consistency in their levels of performance; S1 ranked first on all the measures, and some of the others (S6, S7, and S9) maintained fairly constant levels of performance. Others improved their relative positions as they went on (S2, S5, and S8), others did more poorly (S3) and others (S4) varied from task to task.

For the first graphics task, the students developed a set of depth-first strategies, decomposing the problem into identifiable components such as the body of the house, the roof, and so on, and focusing on programming these components one by one until they had constructed the whole figure. While most of the students used a mix of all three of the depth-first, component-based strategies, some (S6, S7, and S8) used just two, and one (S2) consistently used the incremental strategy throughout the entire session.

The most frequently employed strategy was the incremental, and it was used by all the students. The modular strategy was used the least frequently; it was used either at the outset of the session or after a reorganization of the program, for instance when a superprocedure was written midway through the session. All the students used a form of reactive debugging in which they ran procedures and then repaired them when there was a discrepancy between the output and their intentions. Some students (S4, S5, S7, and S9) used this strategy exclusively, while the others made some appreciable use of anticipatory search and occasionally some debugging tactics.

The students approached the second task, at least initially, in a fundamentally different manner than they had the first. Rather than breaking the problem down into identifiable components, they began by constructing a superprocedure with most of the components included within it. The superprocedure might include, for instance, a procedure for getting the player's name, instructions for generating the random number, and then a recursive procedure for getting the guesses and determining whether or not they were correct. Four (S1, S2, S6, and S7) of the students used breadth-first refinement strategies (S2 began with a nominal strategy but soon switched), two (S4 and S8) employed stratification strategies and three (S3, S5, and S9) used sequential strategies.

The depth-first strategies were used in the context of the higher level strategies, the incremental strategy being the most frequently employed. Only one student (S3) used the modular strategy in the context of this task, and he used it in the context of reorganizing his program. The most frequently used subsidiary strategies was a form of symptomatic debugging in which error messages were taken as clues to the source of problems in procedures. Some debugging tactics were also used. Anticipatory search strategies were used rarely.

No new strategies appeared in the third task. Most of the students began with one of the top-down strategies. The students who had used breadth-first

strategies in the second task switched to the nominal or stratification strategies in this task, perhaps in recognition of the greater complexity of programming the Hangman game. Two (S3 and S5) of the three who used the sequential strategy on the second task abandoned it for more sophisticated strategies; the other (S9) reverted to using only depth-first strategies for the third task. The incremental strategy was the most frequently employed depth-first strategy, although some (S5, S6. and S4) showed preferences for the refinement and modular strategies. Reactive and symptomatic debugging were the most frequently used subsidiary strategies, although there was some use of anticipatory search, especially in the programming of the graphics components of the task.

What characterized the third task most fundamentally was the integration of the graphics and interactive, word-and-list components which were found separately in the two previous tasks. Some students (S2, S2, S4, and S5) took approaches which were consistent with the word-and-list context of the third task, even when programming the graphic components. However, others reverted to the approaches that they had used in the first task in programming the graphics portion of the task; for instance, some drew the hanged man as a static display, as if drawing the house and playhouse, only later linking it with the cycle for guessing the word.

In summary, across the three sessions, all the students used top-down strategy and most of them, at some point, employed the more sophisticated breadth-first and stratification strategies. Over the three sessions, every students used each of the depth-first strategies at least once. Debugging was the predominant method used to develop knowledge within the task environment, although anticipatory search was used especially in programming the graphics.

Observations and discussion

These results and close analysis of the representations of the programming process provide the basis for observations about problem solving in this environment and some general speculations about the nature of goal-directed action. These observations and speculations focus on the way in which knowledge is developed and deployed, the relationship between expertise and strategy use, and the evolution of goal representations in the course of problem solution.

Knowledge is not simply deployed in problem solving, but it is also developed within the process. The development of knowledge is dependent on the feedback that is received as a result of action; in the LOGO task environment, that feedback is received largely through experimentation in the trial space. Although students were able to retrieve existing knowledge for solutions of problems, much of the knowledge they required to complete the task had to be developed within the task environment itself. This knowledge was developed through cyclical search through the two problem spaces, such as in the construct-test-modify cycle so often observed within the protocols. In virtue of the structure of this task environment, therefore, knowledge becomes

very much an interactive accomplishment, both in the sense of the interaction between the student and the task and in the sense of the interplay between the two problem spaces which characterizes the search process.

Within the literature on the programming process, the assumption is made the kinds of strategies used should be a function of the expertise of the programmer. In general, the literature suggests that top-down, breadth-first strategies are used by experts whereas the depth-first and bottom-up strategies are used by novices. However, the types of strategies that these students used seemed to be sensitive to task demands and, although the more skilled students used the more sophisticated strategies, the less skilled used them as well, although clearly to less effect.

Strategy use, rather than being an outcome of the programmer's level of knowledge, seemed to be sensitive to task demands. The first two tasks differ in terms of their demands and the way in which the students approached these tasks appeared to differ as a result. In the case of the first task, to draw a picture of a house and playhouse, it is possible to simply join instructions together in a rather unstructured way. The separate components need not interact, they only have to be joined together. The kinds of depth-first strategies that the students developed were quite sufficient to deal with the demands of this task.

By contrast, an interactive game is a much more integrated and complex structure. To get a game to work, there has to be a definite sequence of instructions and the functioning of one component depends on the functioning of others. The students appeared to respond to the demands of this task by constructing more integrated programs using strategies that involved a top-down view and, in some cases, breath-first development of the components.

It is entirely possible that there would have been very different results if the kind of word-and-list programming required in the second task had been taught before graphics programming. However, some of the students reverted to the kind of programming they had done in the first task when programming the graphic components in the third task. It would seem, then, that certain strategies were viewed as appropriate for certain kinds of problems based on the students' previous experiences with those strategies.

Even the subsidiary strategies were task sensitive to some degree. For instance, the anticipatory strategies, which were used in the first task, were largely abandoned in the context of the interactive game programs, although they were used in programming the graphics of the third task. The students appeared to know how to experiment in the trial space to gather information when dealing with the graphics components, but were unable to use this strategy when it would have required detaching highly interdependent components of the game programs.

Clearly all of these students were novices, yet the fact that they all used a variety of strategies, some of which could be considered relatively

sophisticated, argues against a close linkage between strategy use and levels of expertise. Often students used strategies which required more sophistication than they seemed to possess. To use a refinement strategy effectively, for instance, it is necessary to plan out the whole component and have a good idea of how the various instructions work. Some students appeared to have the requisite knowledge to carry this strategy out effectively, while others who used this strategy did not. Thus, the level of knowledge of the programmer would not seem to dictate or preclude the use of any given strategy, although it might have a very decided impact on how effective that use is.

The last point pertains to the nature of the goal representations that direct the problem solving process. Inspection of the problem behavior graphs suggests that these representations are highly complex and vary in their generality and inclusiveness, that they are not always clearly articulated, and that they evolve and are constructed in the process of problem solving.

The goal representations operate at the level of episodes, and a single episode may include a variety of disparate functions. For instance, within a debugging cycle for the High-Low game, for instance, several bugs may be revealed by a test of the program. The programmer may subsequently debug the procedure to get the player's name and also debug the procedure which generates the random number, and then the programmer will see whether or not these procedures work in the context of the program. In this case, it would appear that it is not one goal but several which control the behaviour until feedback is received.

However, these apparently disparate goals within an episode are not unrelated; for instance, the functions of getting the player's name and generating a random number may be related in virtue of their giving values to variables used in the game or in virtue of some even higher level superordinate function. In other words, what gives the episode unity are the higher level functional connections among these apparently disparate functions. This does not mean that the programmer need be explicitly aware of all the functional relationships involved, however. At a given point, a programmer may have a sense of how the program works and how the actions he or she takes might make it work better, and that may suffice as the programmer's goal representation. Thus, the goal representations which direct the programming process may vary from one moment to the next in their level of inclusiveness and generality and in how clear and well-articulated they are.

Unlike well-structured tasks, the goals with which the programmer starts are not necessarily the goals which end up being achieved; goals will change as obstacles and opportunities present themselves along the search path. In other words, the search process is not merely a process in which goals are set, acted upon, and evaluated, but a process in which goals themselves are constructed and articulated. And the LOGO task environment supports and facilitates this formation of goal representations. The LOGO editor provides an external memory device which supports the programmer's representation of

the program by allowing review of the procedures at any point in the development of the program. Feedback from invoked procedures supports the evolution of goal representations by concretizing them and providing a basis for their further modification and articulation. In essence, then, action within the task environment supports the formation of goals.

Actions are clearly goal-directed, but not necessarily in the sense envisioned in accounts which depend on goals being well articulated. Problem solvers have motivations and intentions, which may or may not lead to clear goal representations, but nevertheless do give direction and momentum (Lewin, 1935) to their actions. The situation of these programmers was often analogous to the situation of a hungry man hunting in a grocery store without a shopping list. The man looks at a variety of different foods, weighing whether this or that one will satisfy the hunger. When asked by a clerk what he is looking for, he replies, "I'll know when I find it." The process of hunting narrows down the range of alternatives and defines the desired object which will satisfy the hunger. The molar activity, "finding something to eat," has a vague goal, obtaining something edible, which becomes delineated in the process of hunting and fully so only when a particular food is seized upon. Action is goal-directed but not goal-driven; what drives the process are motivations and intentions which do not necessarily have a definite object or clear representation. In tasks such as those studied here, a need-system to complete the task, to please an adult, or to prove oneself capable may suffice to drive the process and provide the basis for constructing goals through action.

Thus, goal representations are not static but undergo formation and articulation through action and the feedback received as part of the search cycle. The picture of the process which emerges is of dynamically evolving networks of goals of varying degrees of generality, inclusiveness, and articulation, with different time frames for their achievement (Valsiner, 1987), both determining action and being determined by the results of action.

Implications

The examination of these representations suggests revisions in models of goal-directed behavior which have been based on well-structured tasks. In addition to the theoretical implications of this analysis, there are more concrete implications which are particularly relevant to LOGO programming but may have relevance for other computer-based learning environments.

Within the LOGO community, much emphasis has been placed on teaching planning, the primarily rationale for which has been that this is a possibly transferable skill. However, while clarity of goals at the outset of the problem solving process may be advantageous, the kind of tentative approach to problem solving evident in the records in this study, with poorly articulated goals which only become clarified within the problem solving process, may be natural and inevitable. What may be most significant about this task environment is that it is so supportive of the process of goal articulation.

Given the highly dubious expectations for transfer of skills, excessive focus on planning may be unnecessary and even counterproductive. What this study suggests is the importance of engineering supportive computer-based learning environments which enable students to cope with highly complex tasks.

References

Barker, R., & Wright, H. F. (1971). Midwest and Its Children. Hamsden, CT: Archon Books.

Brenner, M.; Ginsberg, G. P.; & von Cranach, M. (1985). Introduction. In G. P Ginsburg, M. Brenner, & M. von Cranach, Discovery Strategies in the Psychology of Action (pp. 1-18). Hillsdale, NJ: Lawrence Earlbaum Associates.

Broadbent, D. E. (1985). Multiple goals and flexible procedures in the design of work. In M. Frese & J. Sabini (Eds.), Goal Directed Behavior: The Concept of Action in Psychology (pp. 285-294). Hillsdale, NJ: Lawrence Earlbaum Associates.

Brown, J. S. & Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. In D. Sleeman & J. S. Brown (Eds.), Intelligent tutoring systems (pp. 157-183). London: Academic Press.

Card, S. K., Moran, T. P., & Newell, A. (1983). The psychology of human-computer interaction. Hillsdale, NJ: Lawrence Erlbaum Associates.

Carver, S. M. (1987). Transfer of LOGO debugging skills: Analysis, instruction, and assessment. (Doctoral dissertation, Carnegie-Mellon University, 1986). Dissertation Abstracts International, 48, 282B.

Carver, S. M. & Klahr, D. (1986). Assessing children's debugging skills with a formal model. Journal of Educational Computing Research, 2(4), 487-525.

Clements, D.H. (1985). Research on Logo in education: Is the turtle slow but steady, or not even in the race? Computers in the Schools. 2, 55-71.

Ericsson, K.A. & Simon, H.A. (1984). Protocol Analysis. Cambridge, MA: MIT Press.

Klahr, D. & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. Cognitive Psychology, 20, 362-404.

Kowalski, B. & VanLehn, K. (1988). Cirrus: Inducing subject models from protocol data. Program of the Tenth Annual Conference of the Cognitive Science Society. Hillsdale, NJ: Lawrence Erlaum Association.

Lewin, K. (1935). A dynamic theory of personality. New York: McGraw-Hill.

McAllister, A. (1989). Manual for &RECORDER and &PRINTREC. Unpublished manuscript.

McAllister, A. (1990). Towards modeling the programming process. Unpublished manuscript.

McAllister, A. (1991). An analysis of problem solving strategies in LOGO programming using partially automated techniques. Paper presented at the annual meeting of the American Educational Research Association, Chicago, Illinois, April 3, 1991.

McAllister, A. (1992). Interpreting the programming process: A study of the problem solving strategies of young gifted students learning LOGO. (Doctoral dissertation, York University).

Newell, A. & Simon, H.A. (1972). Human Problem Solving. Englewood Cliffs, NJ: Prentice Hall.

Nisbett, R. E., Wilson, T. D. (1972). Telling more than we can know: Verbal reports on mental processes. Psychological Review, 84, 231-259.

Ohlsson, S. & Langley, P. (1985). Identifying solution paths in cognitive diagnosis. Pittsburgh, PA: Robotics Institute, Carnegie-Mellon University.

Praetorius, N. & Duncan, K. D. (1988). Verbal reports: A problem in research design. In L. P. Goodstein, S. E. Olsen, and H. B. Anderson (Eds.), Tasks, errors, and mental models (pp. 293-314). London: Taylor and Francis.

Smith, J. B; Smith, D. K. & Kupstas, E. (1991). Automated Protocol Analysis: Tools and Methodology, Technical Report TR91-034. Chapel Hill, NC: Department of Computer Science, University of North Carolina at Chapel Hill.

Valsiner, J. (1987). Culture and the Development of Children's Action. New York: John Wiley and Son.

VanLehn, K. & Garlick, S. (1987). Cirrus: An automated protocol analysis tool. In P. Langley (Ed.), Proceedings of the Fourth Machine Learning Workshop (pp. 205-217). Los Altos, CA: Morgan-Kaufman.

Vygotsky, L. (1984). Mind in society. Cambridge, Mass.: Harvard University Press.

Waterman, D. A. & Newell, A. (1972). Preliminary results with a system for automatic protocol analysis. CIP Working Paper #211. Pittsburgh, PA: Department of Psychology and Computer Science, Carnegie-Mellon University.

Appendix A: Partially Automated Process Analysis

Partially automated process analysis (PAPA for short) is the name applied to the method of recording the programming process and developing a representation of it in the form of a problem behavior graph. The following pages illustrate the stages that are involved in collecting and analyzing the programming process.

```
PRS5P1.REC S5P1.REC [S5] [2 - 9 - 1989]
DE: [TO HOUSE] [] [[136] [275]]
ED: [ED] [] [[30] [74]]
ED: [ED "HOUSE] [[[TO HOUSE] [[SIDE1] [REPEAT 4 [FD :SIDE1 RT 90]]]]] [[193] [789]]
IM: [CS] [[74] [16]]
DE: [TO DOOR] [[RT 90] [FD :SIDE1 / 2] [FD :SIDE1 / 3 RT 90] [FD :SIDE1 / 4 RT 90]] [[466][2161]]
ER: [ED :DOOR] [36 [DOOR HAS NO VALUE] @GET.PROC.NAMES [RUN :@ACTION] RUN DOOR] [[58]  [6]]
ED: [ED "DOOR] [[[TO DOOR] [[SIDE1] [RT 90] [FD :SIDE1 / 2 LT 90]]] [[TO RECTANGLE][[SIDE1]
[REPEAT 2 [FD :SIDE1 / 3 RT 90 FD :SIDE1 / 4]]]]] [[256] [2852]]
ED: [ED "HOUSE] [[[TO HOUSE] [[SIDE1] [REPEAT 4 [FD :SIDE1 RT 90]]]]] [[379] [184]]
ED: [ED "HOUSE] [[[TO HOUSE1] [[SIDE1] [REPEAT 4 [FD :SIDE1 RT 90]]]]] [[254] [130]]
ER: [TO HOUSE :SIDE1] [1 [HOUSE IS ALREADY DEFINED] @CONDITIONS.MODES [IF FIRST :@ACTION
="TO [RUN :@ACTION @OP.DEFINITION :@ACTION STOP]] TO HOUSE] [[343] [5]]
ED: [ED "HOUSE1] [[[TO HOUSE1] [[SIDE1] [HOUSE :SIDE1] [DOOR :SIDE1] [RECTANGLE :SIDE1]]]]
        [[600] [1086]]
IM: [HOUSE1 50] [] [[241] [31]]
ED: [ED "RECTANGLE] [[[TO RECTANGLE] [[SIDE1]
        [REPEAT 2 [FD :SIDE1 / 3 RT 90 FD :SIDE1 / 4 RT 90]]]]] [[264] [882]]
IM: [CS] [[40] [4]]
IM: [HOUSE1 50] [] [[88] [14]]
```

Figure A1: Raw record generated by the record generator

The recording program (RECORDER) records the actions taken by the subject, and these actions are classified by modes. These modes are:

1) the immediate mode, which is the mode in which instructions are issued and immediately interpreted;
2) the definition mode, initiated by typing "TO" in the immediate mode, which is used to define procedures;
3) the edit mode, which is the full-screen editor in which procedures are constructed;
4) the error mode, which is "entered" whenever an error occurs;
5) load and save modes are entered whenever procedures are loaded or saved; and
6) restart mode, which occurs when the program crashes and has to be restarted.

Each line represents a complete action; for instance, in the immediate mode, it is the complete action ended with a carriage return, in the define mode, when the definition is completed, and in the edit mode when the editor is exited. The first element on the line classifies the action by mode. What follows is the elements necessary for specifying action. The element following the mode descriptor is the action which initiated entrance into the mode. For the define and edit mode actions, the procedure definitions are listed. The last element in the line is the time stamp information; in tenths of a second, the time from the previous action and time to completion of the action.

```
{1} (13.6 / 27.5 / 41.1 / 0:00:41) DE: TO HOUSE
TO HOUSE
>END

{2} (3 / 7.4 / 10.4 / 0:00:51) ED: ED

{3} (19.3 / 78.9 / 98.2 / 0:02:29) ED: ED "HOUSE
TO HOUSE :SIDE1
REPEAT 4 [FD :SIDE1 RT 90]
END

{4} (7.4 / 1.6 / 9 / 0:02:38) IM: CS

{5} (46.6 / 216.1 / 262.7 / 0:07:01) DE: TO DOOR
TO DOOR
>RT 90
>FD :SIDE1 / 2
>FD :SIDE1 / 3 RT 90
>FD :SIDE1 / 4 RT 90
>END

{6} (5.8 / 0.6 / 6.4 / 0:07:07) ER: ED :DOOR
36 DOOR HAS NO VALUE

{7} (25.6 / 285.2 / 310.8 / 0:12:18) ED: ED "DOOR
TO DOOR :SIDE1
RT 90
FD :SIDE1 / 2 LT 90
END
TO RECTANGLE :SIDE1
REPEAT 2 [FD :SIDE1 / 3 RT 90 FD :SIDE1 / 4]
END

{8} (37.9 / 18.4 / 56.3 / 0:13:14) ED: ED "HOUSE
TO HOUSE :SIDE1
REPEAT 4 [FD :SIDE1 RT 90]
END

{9} (25.4 / 13 / 38.4 / 0:13:53) ED: ED "HOUSE
TO HOUSE1 :SIDE1
REPEAT 4 [FD :SIDE1 RT 90]
END

{10} (34.3 / 0.5 / 34.8 / 0:14:28) ER: TO HOUSE :SIDE1
1 HOUSE IS ALREADY DEFINED

{11} (60 / 108.6 / 168.6 / 0:17:16) ED: ED "HOUSE1
TO HOUSE1 :SIDE1
HOUSE :SIDE1
DOOR :SIDE1
RECTANGLE :SIDE1
```

Figure A2: Record printed by the record generator

An option of the analysis program (ANALYZE) is a program which puts the printed record into a readable form. The lines are numbered. The time information is then listed and includes the time from the previous action, the time to completion of the action, the total time for the action, and the time from the start of the recording of the session. The initiating action follows (bold has been added), and what was done in the mode is listed on the lines below.

```
(1) (13.6 / 27.5 / 41.1 / 0:00:41) DE: TO HOUSE
TO HOUSE
>END
new units:

(2) (3 / 7.4 / 10.4 / 0:00:51) ED: ED

(3) (19.3 / 78.9 / 98.2 / 0:02:29) ED: ED "HOUSE
*TO HOUSE :SIDE1
*1 REPEAT 4 [FD :SIDE1 RT 90]
END
old units:
new units: [[REPEAT 4] 1] [[FD :SIDE1] 2] [[RT 90] 3]
3 inserts: [[REPEAT 4] 1] [[FD :SIDE1] 2] [[RT 90] 3]

(4) (7.4 / 1.6 / 9 / 0:02:38) IM: CS

(5) (46.6 / 216.1 / 262.7 / 0:07:01) DE: TO DOOR
TO DOOR
1 >RT 90
2 >FD :SIDE1 / 2
3 >FD :SIDE1 / 3 RT 90
4 >FD :SIDE1 / 4 RT 90
>END
new units: [[RT 90] 1] [[FD :SIDE1] 2] [[/ 2] 3] [[FD :SIDE1] 4] [[/ 3] 5] [[RT 90] 6] [[FD :SIDE1] 7] [[/
4] 8]     [[RT 90] 9]
9 inserts:  [[RT 90] 1] [[FD :SIDE1] 2] [[/ 2] 3] [[FD :SIDE1] 4] [[/ 3] 5] [[RT 90] 6] [[FD :SIDE1] 7] [[/
4] 8]     [[RT 90] 9]

(6) (5.8 / 0.6 / 6.4 / 0:07:07) ER: ED :DOOR
36 DOOR HAS NO VALUE

(7) (25.6 / 285.2 / 310.8 / 0:12:18) ED: ED "DOOR
*TO DOOR :SIDE1
=1 RT 90
*2 FD :SIDE1 / 2 LT 90
END
old units: [[RT 90] 1] [[FD :SIDE1] 2] [[/ 2] 3] [[FD :SIDE1] 4] [[/ 3] 5] [[RT 90] 6] [[FD :SIDE1] 7] [[/ 4]
8][[RT 90] 9]
new units: [[RT 90] 1] [[FD :SIDE1] 2] [[/ 2] 3] [[LT 90] 4]
6 deletes: [[FD :SIDE1] 4] [[/ 3] 5] [[RT 90] 6] [[FD :SIDE1] 7] [[/ 4] 8] [[RT 90] 9]
1 inserts: [[LT 90] 4]
TO RECTANGLE :SIDE1
1 REPEAT 2 [FD :SIDE1 / 3 RT 90 FD :SIDE1 / 4]
END
new units: [[REPEAT 2] 1] [[FD :SIDE1] 2] [[/ 3] 3] [[RT 90] 4] [[FD :SIDE1] 5] [[/ 4] 6]
5 inserts: [[REPEAT 2] 1] [[FD :SIDE1] 2] [[/ 3] 3] [[RT 90] 4] [[FD :SIDE1] 5] [[/ 4] 6]

(8) (37.9 / 18.4 / 56.3 / 0:13:14) ED: ED "HOUSE
Unchanged: HOUSE
```

Figure A3: Record generated by the encoder

Another option of the analysis program (ANALYZE) is designed to facilitate manual checking and elaboration of the automated encoding. The lines of a procedure are numbered to facilitate comparisons with the previous version. An asterix indicates a change in the line, an equal sign indicates the line remains the same. Old and new instructional units are listed. Each unit (defined as a LOGO primitive and its inputs) is listed and numbered, and then the differences between the old and new version of the procedure are given in terms of inserts and deletes.

```
FILENAME: B:KSP1.REC
[define HOUSE] [] [def] [1] [HOUSE] []
[] [] [app] [1] [HOUSE] [] [41.1]
[] [] [unchanged] [2] [] [][10.4]
[] [] [] [3] [HOUSE] [][98.2]
[] [] [invoke1] [4] [CS] [] [9]
[define DOOR] [] [def] [5] [DOOR] []
[] [] [app] [5] [DOOR] [] [262.7]
[] [] [error] [6] [ED :DOOR] [DOOR HAS NO VALUE] [6.4]
[] [] [] [7] [DOOR] []
[define RECTANGLE] [] [def] [7] [RECTANGLE] []
[] [] [app] [7] [RECTANGLE] [] [310.8]
[] [] [unchanged] [8] [HOUSE] [][56.3]
[define HOUSE1] [] [def] [9] [HOUSE1] []
[] [] [app] [9] [HOUSE1] [] [38.4]
[] [] [error] [10] [TO HOUSE :SIDE1] [HOUSE IS ALREADY DEFINED] [34.8]
[] [] [] [11] [HOUSE1] [][168.6]
[] [] [invoke] [12] [HOUSE1 50] [] [27.2]
[] [] [] [13] [RECTANGLE] [][114.6]
[] [] [invoke] [14] [CS] [] [4.4]
[] [] [invoke] [15] [HOUSE1 50] [] [10.2]
```

Figure A4: Empty slots produced by the functional record generator

The slot record can be generated by the analysis program (ANALYZE); it consists of a series of lines corresponding to the actions in the record. Operators are identified in the encoding process. This slot record is used manually the investigator to group and identify the functions of the operators. In grouping operators according to their functions, additional lines are added as required. The description of the function goes in the first slot, and the description of its superordinate function in the second slot. The third slot is to be filled with the type of operator (i.e., append, insert, delete, invoke, unchanged, error, etc.). The line number, the initiating action, comments or error messages, and the total time the action took follow.

```
FILENAME: B:SSP1.REC
[whole] [] [] [] [] []
[square] [whole] [] [] [] []
[define HOUSE] [square] [def] [1] [HOUSE] [] [41.1]
[examine HOUSE] [square] [unchanged] [2] [HOUSE] [] [10.4]
[use variables in command line of HOUSE] [square] [ins w/in] [3] [HOUSE] []
[make square] [square] [app] [3] [HOUSE] [][98.2]
[clearscreen] [square] [invoke] [4] [CS] [] [9]
[door] [whole] [] [] [] []
[define DOOR] [door] [def] [5] [DOOR] []
[position, draw door] [door] [app] [5] [DOOR] [][262.7]
[edit DOOR] [door] [error] [6] [ED :DOOR] [DOOR HAS NO VALUE] [6.4]
[take out door] [position, draw door] [del] [7] [DOOR] []
[turn in to draw door] [position, draw door] [app] [7] [DOOR] []
[rectangle] [door] [] [] [] []
[define RECTANGLE] [rectangle] [def] [7] [RECTANGLE] []
[do rectangle] [rectangle] [app] [7] [RECTANGLE] [] [310.8]
[examine HOUSE] [square] [unchanged] [8] [HOUSE] [] [56.3]
[define HOUSE1] [whole] [def] [9] [HOUSE1] [makes a superprocedure]
[use contents of HOUSE] [whole] [app] [9] [HOUSE1] [] [38.4]
[define HOUSE] [whole] [error] [10] [TO HOUSE :SIDE1] [HOUSE IS ALREADY DEFINED] [34.8]
[empty contents of HOUSE1] [whole] [del] [11] [HOUSE1] []
[make call to HOUSE] [whole] [app] [11] [HOUSE1] []
[make call to DOOR] [whole] [app] [11] [HOUSE1] []
[make call to RECTANGLE] [whole] [app] [11] [HOUSE1] [][168.6]
[test HOUSE1] [whole] [invoke] [12] [HOUSE1 50] [door on side and incomplete] [27.2]
[put in last angle] [whole] [app] [13] [RECTANGLE] [] [114.6]
[test HOUSE1] [whole] [] [] [] []
[clearscreen] [test HOUSE1] [invoke] [14] [CS] [] [4.4]
[test it] [test HOUSE1] [invoke] [15] [HOUSE1 50] [][10.2]
```

Figure A5: Filled slots for generating the functional record

This shows the slots as filled by the investigator. The record begins by designating a top-level function, which is to complete the whole task. The next function is the initial function which is to draw a square with the whole as its superordinate function. Operators are grouped according to their specific functions. The next general function is the door, which also has the whole as its superordinate function. The creation of the superprocedure, HOUSE1, which integrates the door and the square, has the whole as its superordinate function.

```
FILENAME A:SSP1.SL
G: whole
   G: square
      G: define HOUSE ( def ) < 1 > { HOUSE } < 41.1 >
      G: examine HOUSE ( unchanged ) < 2 > { HOUSE } < 10.4 >
      G: use variables in command line of HOUSE ( ins w / in ) < 3 > { HOUSE }
      G: make square ( app ) < 3 > { HOUSE } < 98.2 >
      G: clearscreen ( invoke ) < 4 > { CS } < 9 >
   G: door
      G: define DOOR ( def ) < 5 > { DOOR }
      G: position, draw door ( app ) < 5 > { DOOR } < 262.7 >
      G: edit DOOR ( error ) < 6 > { ED :DOOR } < 6.4 >
           ( DOOR HAS NO VALUE )
      S: position, draw door
         G: take out door ( del ) < 7 > { DOOR }
         G: turn in to draw door ( app ) < 7 > { DOOR }
      G: rectangle
         G: define RECTANGLE ( def ) < 7 > { RECTANGLE }
         G: do rectangle ( app ) < 7 > { RECTANGLE } < 310.8 >
   S: square
      G: examine HOUSE ( unchanged ) < 8 > { HOUSE } < 56.3 >
   G: define HOUSE1 ( def ) < 9 > { HOUSE1 }
        ( makes a superprocedure )
   G: use contents of HOUSE ( app ) < 9 > { HOUSE1 } < 38.4 >
   G: define HOUSE ( error ) < 10 > { TO HOUSE :SIDE1 } < 34.8 >
        ( HOUSE IS ALREADY DEFINED )
   G: empty contents of HOUSE1 ( del ) < 11 > { HOUSE1 }
   G: make call to HOUSE ( app ) < 11 > { HOUSE1 }
   G: make call to DOOR ( app ) < 11 > { HOUSE1 }
   G: make call to RECTANGLE ( app ) < 11 > { HOUSE1 } < 168.6 >
   G: test HOUSE1 ( invoke ) < 12 > { HOUSE1 50 } < 27.2 >
        ( door on side and incomplete )
   G: put in last angle ( app ) < 13 > { RECTANGLE } < 114.6 >
   G: test HOUSE1
      G: clearscreen ( invoke ) < 14 > { CS } < 4.4 >
      G: test it ( invoke ) < 15 > { HOUSE1 50 } < 10.2 >
```

Figure A6: Functional record

Another program (GENERATOR) takes as input the slotted record and prints out a functional record which is indented according to the structure of the functions. "G" represents a new function or goal; all the functions appearing below it to its right are considered subordinate functions. "S" represents a superordinate function which has been previously identified.

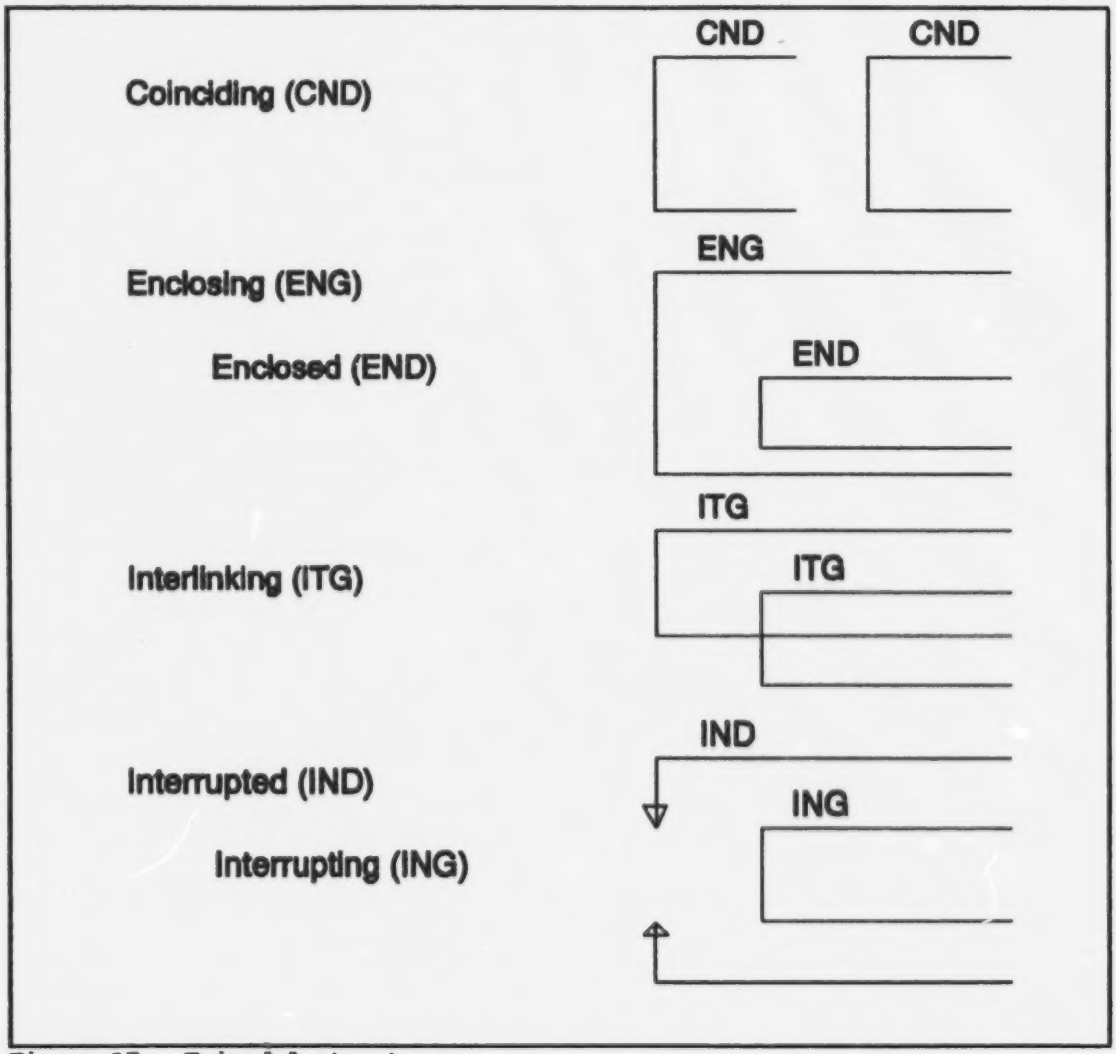

Figure A7: Episodal structures

The technique of episoding specimen descriptions employed by Barker and Wright (1971) is used on the functional record. Barker and Wright's categorizations and representations of episodes are closely followed here. Coinciding episodes occur when different episodes intersect from beginning to end. Enclosing and enclosed episodes occur when a part of an episode intersects with the whole of another. The longer episode is the enclosing episode, the shorter the enclosed. Interlinking episodes occur when one part of an episode intersects with part of another. Interrupting episodes occur when an episode occurs in the context of another episode but has a different direction. Isolated episodes (not illustrated) occur when an episode occurs alone.

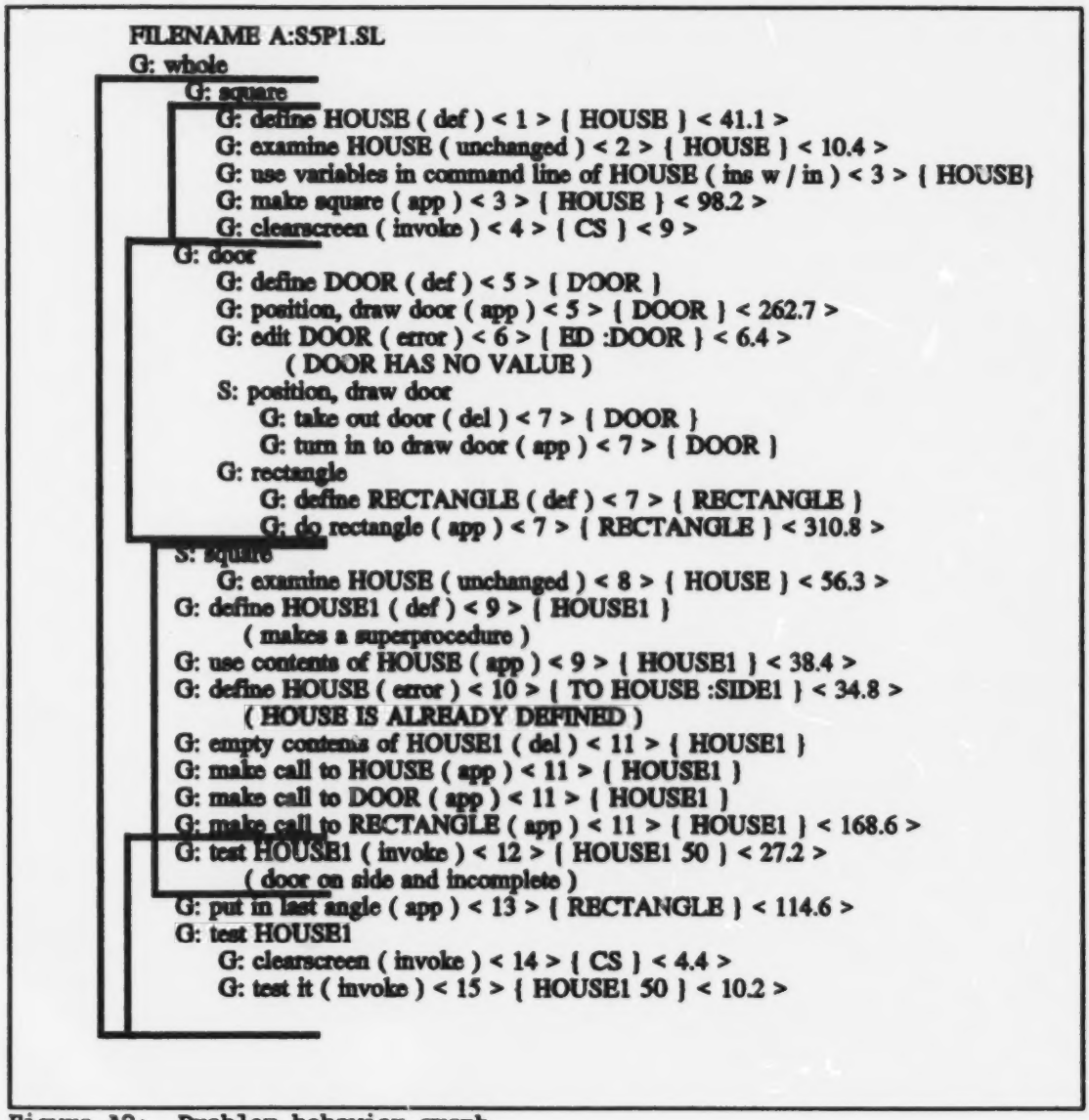

Figure A8: Problem behavior graph

The problem behavior graph is developed by the investigator manually episoding the functional record. A complete episode is represented here which includes drawing a square shape with a door, most of which occurs within the procedural space. The subepisodes consist of actions of developing procedures to draw a square and then to draw a door, integrating them into a superprocedure, and finally the testing and debugging of the superprocedure. This is an example of a modular strategy.

Table 1: Levels of strategies

Strategies	articulation at outset	component development	way trial space employed
top-down			
breadth-first refinement	whole at all levels	refined	
nominal	abstract level only	depth-first	
stratification	sufficient for testing	depth-first as required	
sequential	by order of instructions	depth-first in order	
component-based			
refinement		all at once	
incremental		bit by bit	
modular		by subunits	
subsidiary			
anticipatory search			get information prior to testing
debugging strategies			seek feedback for modifications
tactics for debugging			get information for debugging

Table 2: Results of measures

ID	ranks components	ranks knowledge	1st task strategies	2nd task strategies	3rd task strategies
S1	1, 1, 1	1, 1, 1	ref, inc-mod	bfr inc, ref	strat inc, ref, mod
S2	4, 4, 2	5, 5, 2	inc	nom -> bfr inc, ref	strat inc, mod
S3	2, 7, 8	4, 8, 5	inc, mod, ref	seq inc, mod	strat inc
S4	5, 2, 7	7, 4, 7	ref, mod, inc	strat inc-ref	strat ref
S5	9, 5, 3	2, 7, 3	ref, inc, mod	seq inc	bfr mod-ref
S6	3, 3, 5	6, 3, 4	inc. ref	bfr inc	nom ref-mod
S7	6, 6, 6	8, 6, 9	inc, ref	bfr ref-inc	strat inc, ref, mod
S8	7, 8, 4	3, 2, 8	inc, ref	strat inc	strat inc, mod
S9	8, 9, 9	9, 9, 6	inc, ref, mod	seq inc	inc, ref

Note. ranks components refers to the ranking of the students according to the number of components they were able to implement within the time limits; ranks knowledge refers to the ranking of the students according to the number of items they got correct on the test of knowledge. inc = incremental strategy, ref = refinement strategy, mod = modular strategy, bfr = breadth-first refinement strategy, strat = stratification strategy, nom = nominal strategy, seq = sequential strategy.

Table 3: Intercorrelations among measures

	1st task	1st in-cls	2nd task	2nd in-cls	3rd task	3rd in-cls
1st task	1.00	.44	.41	.34	.30	.27
1st in-cls	.44	1.00	.34	.50	.83*	.55
2nd task	.41	.34	1.00	.65*	.48	.55
2nd in-cls	.34	.50	.65*	1.00	.69*	.24
3rd task	.31	.83*	.48	.68*	1.00	.73*
3rd in-cls	.27	.55	.55	.24	.73*	1.00

Note. The programming tasks are referred as tasks, whereas the in-class measures of programming knowledge are referred to using the abbreviation "in-cls." The asterix indicates a significant correlation ($p > .10$).

House-playhouse

```
TO HOUSE
REPEAT 4 [FD 40 RT 90]
FD 40 RT 30
REPEAT 3 [FD 40 RT 90]
FD 40 RT 30
FD 40 RT 90 FD 15 RT 90
REPEAT 2 [FD 20 LT 90 FD 10 LT 90]
END
```

```
TO HOUSE
SQUARE
TRIANGLE
DOOR
END
```

High-low

```
TO HIGH.LOW
GET.NAME
GENERATE.RANDOM.NUMBER
PLAY.GAME
INFORM.BY.NAME
PRINT.NUMBER.OF.GUESSES
PRINT.GUESSES
END

TO PLAY.GAME
ENTER.GUESS
CHECK.GUESS
IF WON? = "YES [STOP]
PLAY.GAME
END
```

```
Hello, what's your name?  Alan
Enter your guess.  50
     Too low.
Enter your guess.  75
     Too high.
Enter your guess.  62
     Too high.
Enter your guess.  56
     Correct Alan; 4 guesses
Here they are:
     50
     75
     62
     56
```

Hangman

```
TO HANGMAN
ENTER.SECRET.WORD
DRAW.GALLOWS
PLAY.GAME
END

TO PLAY.GAME
ENTER.GUESS
IF IN.WORD? = "YES [GET.POSITION]
IF IN.WORD? = "NO [HANG]
IF ALL.LETTERS? = "YES [WON STOP]
IF ALL.HUNG? = "YES [LOST STOP]
PLAY.GAME
END
```

```
H A N _ M A N
```

Figure 1: The three tasks made simple

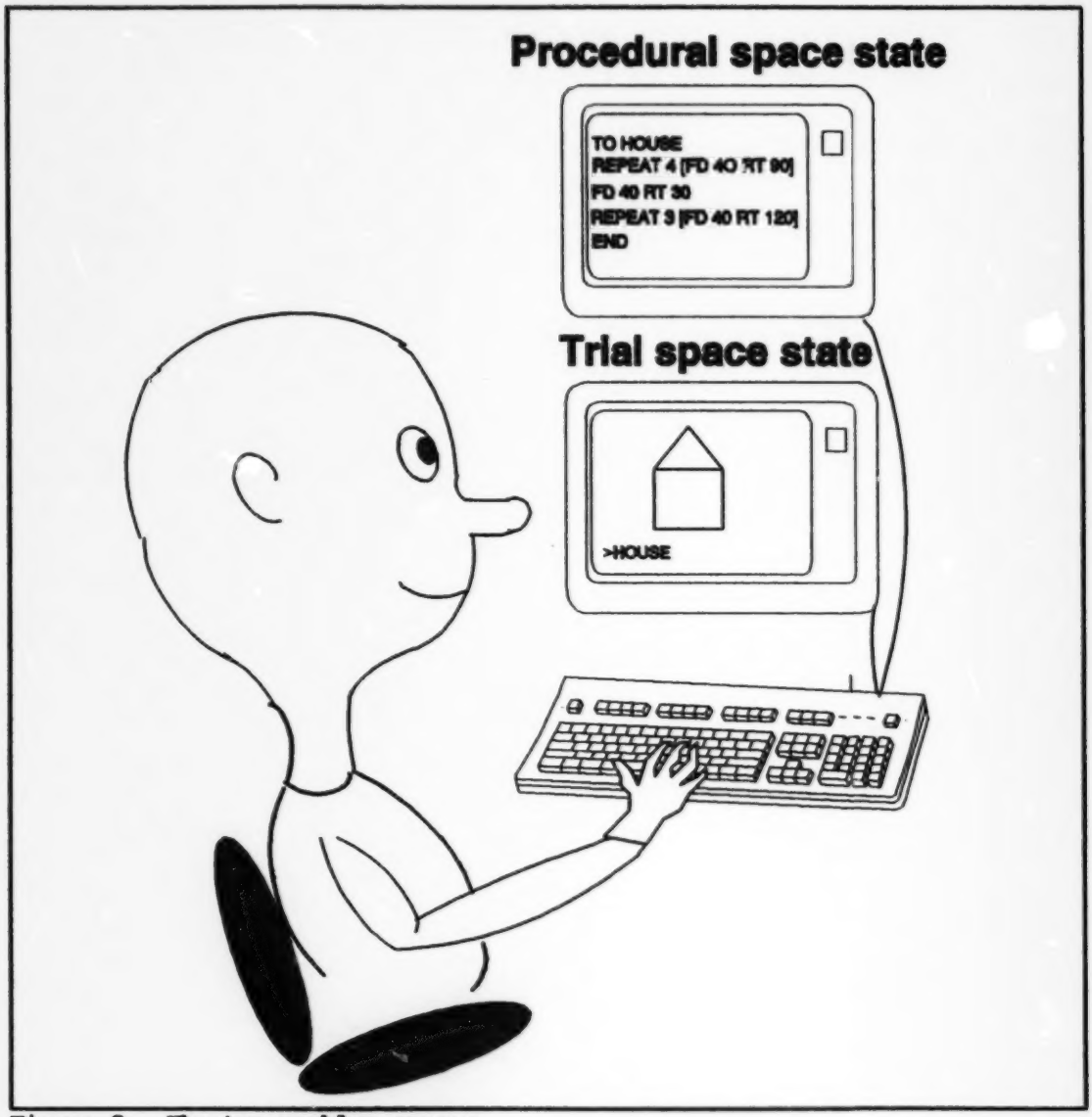

Figure 2: The two problem spaces

Note. This is, of course, a bit of a caricature of the two problem spaces. It is loosely based on a figure appearing in Card, Moran and Newell (1984). Problem spaces and their states are symbolic structures rather than physical states on a computer screens, but their content can be suggested by the interaction of the hairless fellow with the two screens on which are represented a procedure under development in the edit mode and its output in the immediate mode.

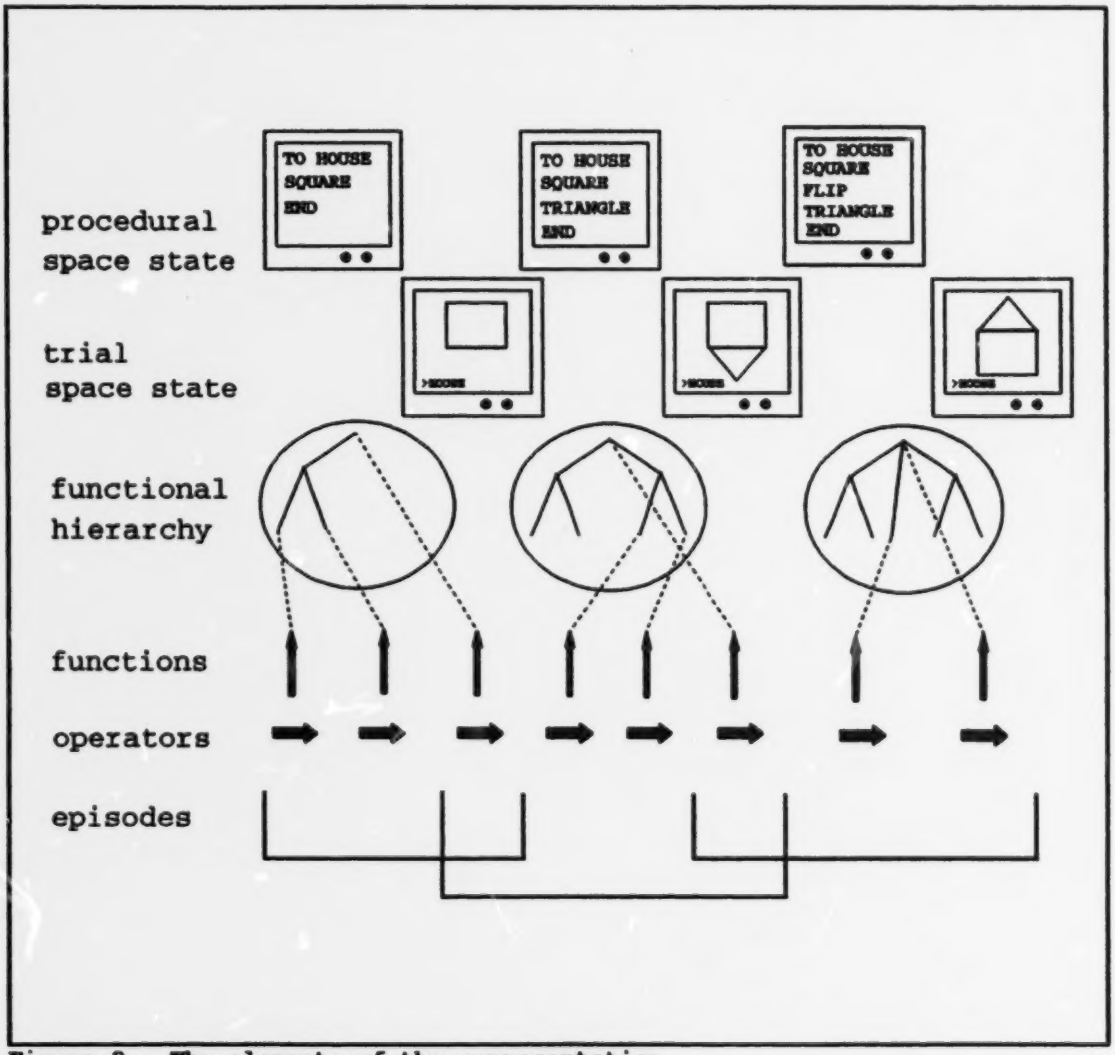

Figure 3: The elements of the representation

Note. The elements involved in the representation of the programming process are pictured here in an abstract form. The screens represent states of the procedural space and the trial space as the program evolves. The operators are indicated as horizontal arrows, with the vertical arrows indicating the functions of the operators which take their place in the functional hierarchies represented in the trees in the circles. The brackets at the bottom show the episodes. Pictured here is a set of interlinked episodes and a cycle in which the procedure is constructed, tested, and revised.

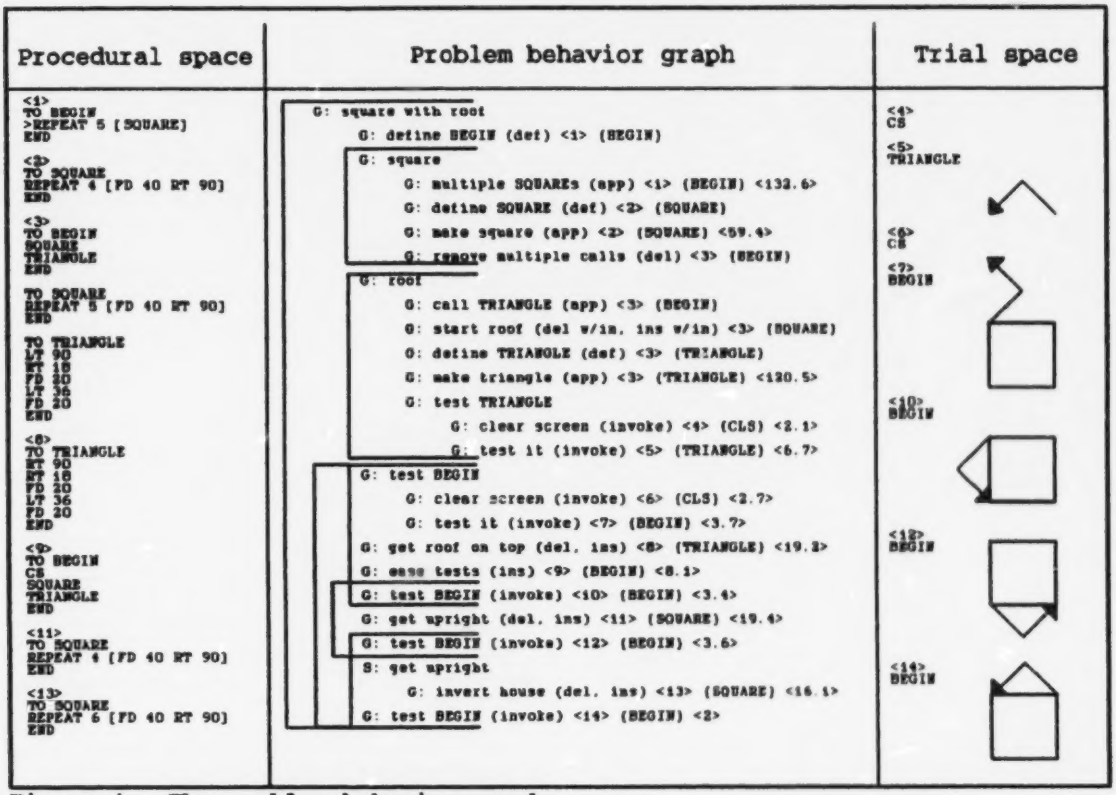

Figure 4: The problem behavior graph

Note. States of the two problem spaces are represented on the left and the right of the problem behavior graph and are keyed to the graph by line numbers in the record. The graph itself is an episoded functional record. The brackets demarcate the episodes. Seven episodes are shown; two enclosing, six enclosed, and three interlinked (see Barker & Wright, 1971). Indentations represent the hierarchical structure of goals and subgoals. "G" stands for goal; a previously set superordinate goal which is not adjacent to the subgoal is indicated by an "S" rather than a "G." The goals are briefly identified. The operators associated with these goals operate on instructional units; such a unit is defined as a LOGO primitive or defined procedure and its inputs. Categories of operators are indicated in parentheses. The procedural space operators are "def" for define, "app" for append, "ins" for insert, and "del" for delete (when a part of an instructional unit is the object of an operator, "w/in" is indicated). Trial space operators are designated as "invoke," and operators that result in errors are indicated simply as "error." The numbers immediately following the operators indicate the line number in the record, and the action or procedure affected are indicated in brackets after the line number. Error messages follow in parenthesis. The time from the last action to the completion of the action is given for the last set of operators for the line. This figure shows a depth-first, component-based modular strategy in which subcomponents of the square with roof are separately defined (SQUARE and TRIANGLE) and then integrated into the whole (BEGIN).

BEST COPY AVAILABLE

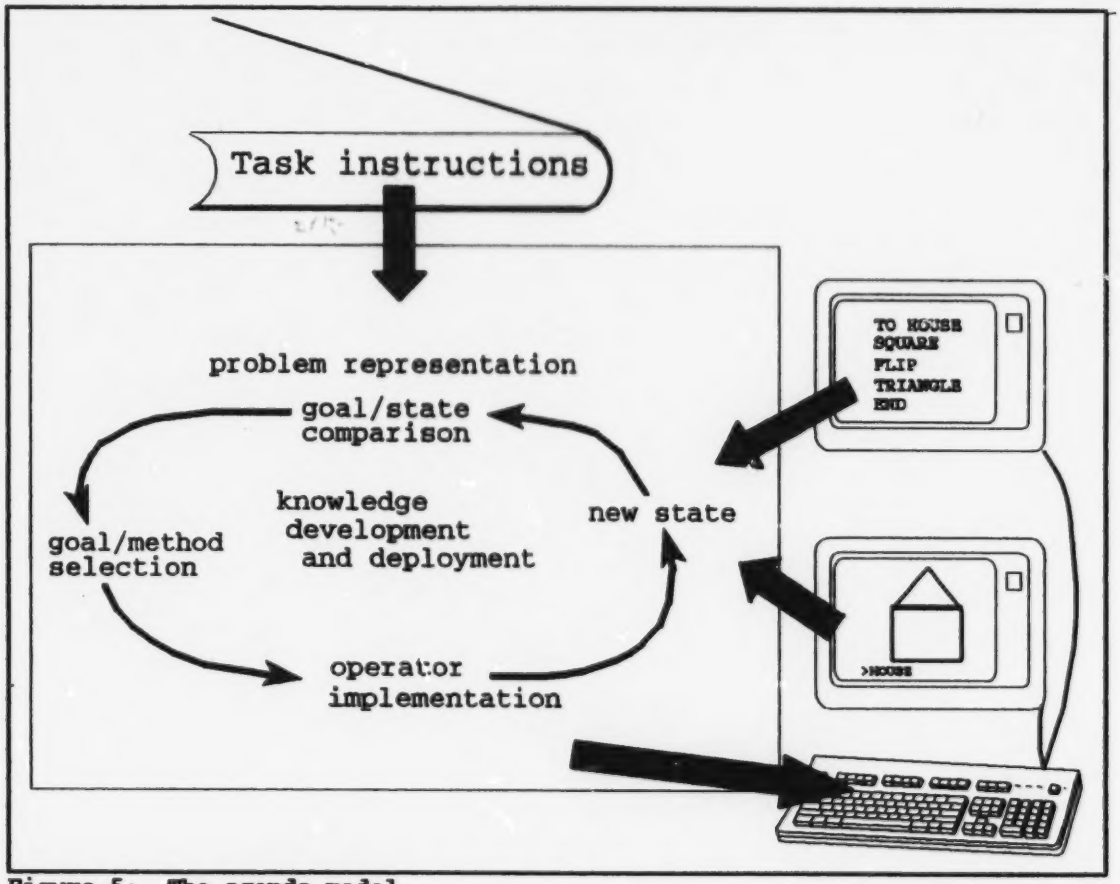

Figure 5: The agenda model

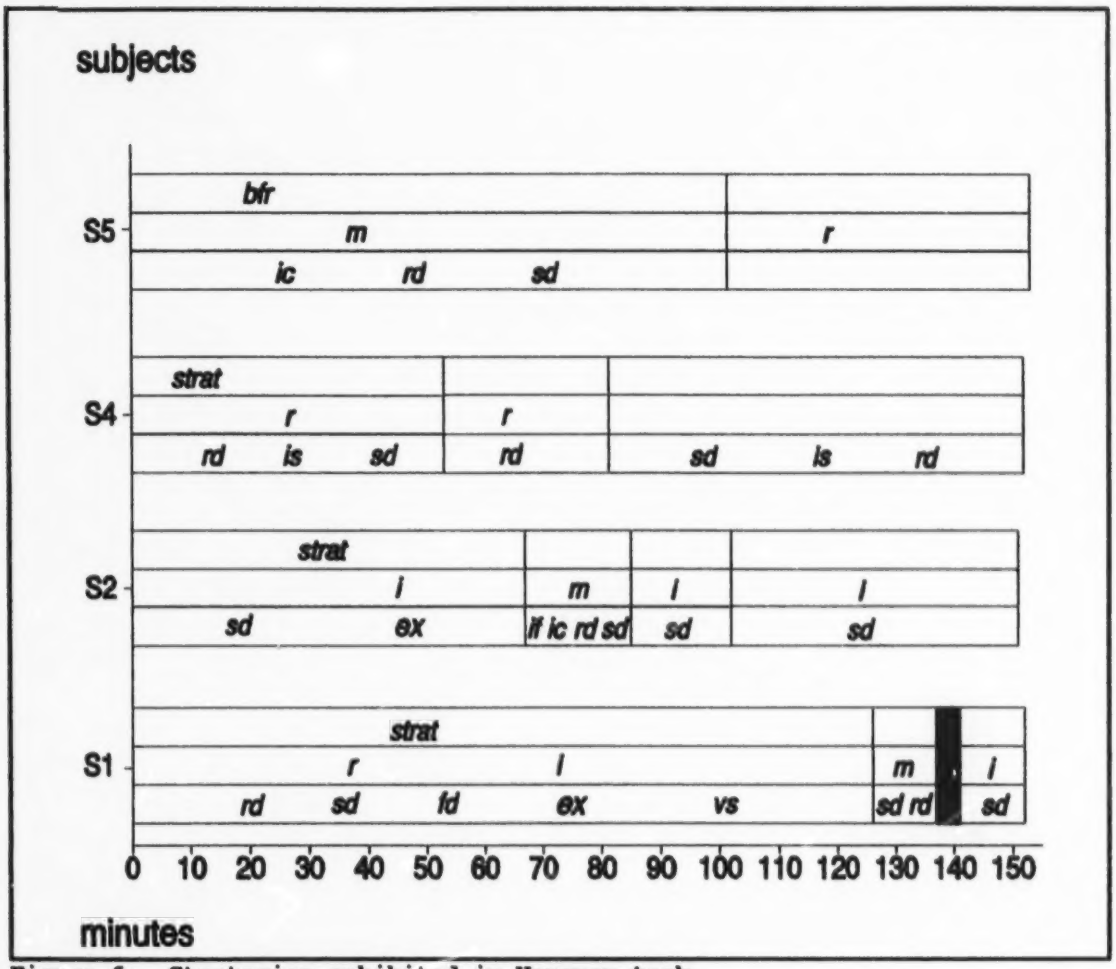

Figure 6: Strategies exhibited in Hangman task

Note. ***bfr*** = breadth-first refinement strategy, ***n*** = nominal, ***strat*** = stratification, ***seq*** = sequential strategy, ***i*** = incremental strategy, ***m*** = modular strategy, ***r*** = refinement strategy, ***ic*** = incorporative search, ***if*** = inferential search, ***is*** = isolation search, ***rd*** = reactive debugging, ***sd*** = symptomatic debugging, ***fd*** = focused debugging, ***ex*** = exploratory debugging, ***vs*** = verification search. The divisions within the bars the time frames of the episodes. Three levels of strategies are represented: The top-down strategies are represented in the top of the bars; the depth-first, component-based strategies are represent in the middle portion of the bars, and the subsidiary strategies are represented in the bottom portion. The darkened portion in the bar for S1 indicates an interruption.